Metaheurísticas

Problema del Aprendizaje de Pesos en Características.

Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy.

UNIVERSIDAD DE GRANADA

E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática. Tercero.
Curso 2018-2019.

Algoritmos 1-NN, Greedy-Relief y Búsqueda Local.

Daniel Bolaños Martínez
76592621-E
danibolanos@correo.ugr.es
Grupo 1 - Martes 17:30h

Índice

1. **Descripción del Problema……...……..1**

2. **Descripción de los Algoritmos empleados en el Problema…….....…….....2-3**
   k-NN…….....2

3. **Pseudocódigo de los Algoritmos…...……...…….....3-5**
   Algoritmos Greedy (Relief)…….....3-4
   Búsqueda Local……...4-5

4. **Descripción en Pseudocódigo de los Algoritmos de Comparación...…….....6**

5. **Procedimiento del Desarrollo de la Práctica y Manual de Usuario....…….....6-7**

6. **Experimentos y Análisis de Resultados……...…….....7-12**

7. **Bibliografía……...…….....12**

1. Descripción del Problema

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos.

La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así se puede formular como:

$$Maximizar\ F(W) = \alpha \cdot tasa\_clas(W) + (1 - \alpha) \cdot tasa\_red(W)$$

Partimos de un vector de datos $T = \{t_1, \ldots, t_n\}$ donde cada dato contiene un conjunto de características o *traits* $\{x_1, \ldots, x_n\}$.

El problema se reduce en calcular un vector de pesos $W = \{w_1, \ldots, w_n\}$ donde el peso $w_i \in [0, 1]$ pondera la característica x_i. A este vector lo denominaremos vector de entrenamiento o aprendizaje y nos permitirá clasificar otro conjunto de datos desconocido al que llamaremos vector de test o validación.

Para aumentar la fiabilidad del proceso usaremos el método *5-fold cross validation* que consiste en crear cinco particiones distintas de datos repartidos equitativamente según su clase. Las particiones las dividiremos de tal forma que dediquemos una a *test* y cuatro a *train*.

Para clasificar los datos, implementamos el algoritmo **k-NN** en su versión con $k = 1$. Este algoritmo asigna a cada dato la clase su vecino más cercano calculado a partir de la *distancia euclídea*.

Calcularemos el vector de pesos con los diferentes algoritmos y usaremos nuestro clasificador **k-NN** para valorar el rendimiento de los resultados. Además es necesario calcular los diferentes porcentajes para cada conjunto de datos valorados. Definiremos:

$$tasa\_clas = 100 \cdot \frac{n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0.2}{n^o\ caracteristicas}$$

Para proceder a los cálculos debemos partir de unos datos normalizados. En esta primera práctica, implementamos un algoritmo **Greedy (Relief)** y uno de **Búsqueda Local** y hacemos ejecuciones sobre un conjunto de datos que contienen los archivos *colposcopy*, *ionosphere*, *texture* todos ellos en el formato elegido *.csv*.

2. Descripción de los Algoritmos empleados en el Problema

Nuestro esquema de representación de la solución, será un vector de pesos W que nos servirá para valorar la bondad de los datos en clasificaciones futuras. El vector será del tipo $W = \{w_1, \ldots, w_n\}$ donde cada $w_i \in [0, 1]$ y cada w_i gradúa el peso asociado a cada característica y pondera su importancia.

Nuestro algoritmo **k-NN** medirá la bondad de los pesos calculados mediante cada algoritmo, recibiendo como parámetros un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculado por el algoritmo seleccionado. El clasificador devolverá un *struct Resultados* que contiene las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo que ha tardado en ejecutarse.

La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. A mayor tasa de clasificación, mejor será el vector de pesos W generado por nuestro algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto al total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

El pseudocódigo del algoritmo **k-NN** es el siguiente:

```
function KNN(train, test, w)
   for i=0, i < tamaño test, i++ do
      pos = nearestNeighbour(train, test[i], w)
      if train[pos].clase = test[i].clase then
          aciertos++
      end if
   end for
   for i=0, i < tamaño w, i++ do
     if w[i] < 0.2 then
       num_w_menor++
     end if
   end for
  tasa_clas = 100.0*(aciertos / tamaño test)
  tasa_red = 100.0*(num_w_menor / tamaño w)
end function
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que estemos valorando. Además, ha sido programado aplicando leave-one-out, ya que cuando se utiliza en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto, porque estamos aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo dato que estamos comparando.

```
function nearestNeighbour(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1
   for i=0, i < tamaño train, i++ do
      // leave-one-out
      if actual != train[i] then
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
      end if
   end for
end function
```

Finalmente utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de características teniendo en cuenta los pesos cuyo valor supere 0.2.

```
function euclideanDistance(v1, v2, w)
   for i=0, i < tamaño v1, i++ do
      if w[i] >= 0.2 then
         dist = dist + w[i]*(v2[i]-v1[i])*(v2[i]-v1[i])
      end if
   end for
end function
```

3. Pseudocódigo de los Algoritmos

3.1. Algoritmo Greedy (Relief)

El algoritmo se basa en incrementar el peso de aquellas características que mejor separan a ejemplos que son enemigos entre sí y reducir el valor del peso en aquellas características que separan ejemplos que son amigos entre sí.

El pseudocódigo del algoritmo **Greedy (Relief)** es el siguiente:

```
function Relief(train, w)
   w = {0,...,0}
   for i=0, i < tamaño train, i++ do
      nearestFriendEnemy(train, train[i], friend, enemy)
      for j=0, j < tamaño w, j++ do
         w[j]= w[j]+|train[i].t[j]-train[i].t[enemy]|
               -|train[i].t[j]-train[i].t[friend]|
      end for
   end for
   w_max = máximo del vector de pesos w
```

```
   for i=0, i < tamaño w, i++ do
      if w[i] < 0 then
         w[i] = 0
      else
         w[i] = w[i] / w_max
      end if
   end for
end function
```

La función *nearestFriendEnemy* calcula el amigo y el enemigo más cercano a un dato dado en función de la *distancia euclídea*. Un dato se considera enemigo si tiene la clase distinta y amigo si tiene la misma clase del dato actual. La función emplea leave-one-out para evitar comparar distancias con el dato actual.

```
function nearestFriendEnemy(train, actual, friend, enemy)
   for i=0, i < tamaño train, i++ do
      if actual != train[i] then
         distancia_actual = euclideanDistance(train[i].t, actual.t)
         if train[i].category = actual.category then
            if distancia_actual < mejor_distancia_a then
               mejor_distancia_a = distancia_actual
               friend = i
            end if
         end if
      else
         if distancia_actual < mejor_distancia_e then
            mejor_distancia_e = distancia_actual
            enemy = i
         end if
      end if
   end for
end function
```

3.2. Búsqueda Local

La búsqueda local implementa una búsqueda de primero el mejor. El vector index nos indica en qué orden se van a modificar las componentes. Modificando así en cada paso una componente aleatoria que no hayamos modificado antes. El vector de pesos W se generará de manera aleatoria con valores entre $[0, 1]$ utilizando una distribución uniforme real.

Para poder generar soluciones nuevas, deberemos modificar/mutar el vector W añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza σ^2, pero este método puede proporcionar soluciones negativas, por lo que debemos de truncar los valores negativos a 0.

El pseudocódigo del algoritmo **Búsqueda Local** es el siguiente:

```
function BL(train, w, semilla)
   w = distribucion_uniforme(0,1)
   index = {0,...,w.size()}

   mezcla los valores de index

   // clasifica el vector de pesos w con KNN y
   // calcula su agregado como tasa de evaluación
   antiguo = KNN(train, train, w)
   agr_ant = agregado(antiguo.clas,antiguo.red)

   while iter < MAX_ITER and neighbour < tamaño w*MAX_NEIGHBOUR do
      aux = index[iter % w.size()]
      w_mut = w
      w_mut[aux] = w_mut[aux] + normal(generator)
      truncar el vector de pesos mutado
      //clasifica el vector de pesos mutado
      // y calcula su agregado
      nuevo = KNN(train, train, w_mut)
      agr_new = agregado(nuevo.clas,nuevo.red)
      iter++
      if agr_new > agr_ant then
         w = w_mut
         agr_ant = agr_new
         neighbour = 0
      else
         neighbour++
      end if
      if iter % tamaño w = 0 then
         mezcla los valores de index
      end if
   end while
end function
```

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso y utiliza este resultado para evaluar la bondad de la solución obtenida. En este caso usamos alpha=0.5.

```
function agregado(t_clas, t_red)
  alpha*t_clas+(1.0-alpha)*t_red
end function
```

4. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar los datos obtenidos para cada algoritmo es siempre el mismo y se realiza en la función *ejecutar* para cada algoritmo programado.

Primero obtenemos los pesos usando el algoritmo a comparar y seguidamente clasificamos con el **k-NN** los datos de train y test sobre ese vector de pesos obtenido. Por último devolvemos en un *struct Resultados* las tasas de clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

```
function ejecutarAlgoritmo(particion, i)
   test = toma la partición i
   train = toma la suma de datos de las particiones !=
   w = Algoritmo(train, w)
   resultados = KNN(train, test, w)
end function
```

5. Procedimiento del Desarrollo de la Práctica y Manual de Usuario

La práctica ha sido programada en el lenguaje C++. Para el desarrollo de la práctica, primero he necesitado leer los archivos que nos proporcionan los datos, para ello, he optado por pasar los archivos .arff a .csv, un formato que al menos para mí, es más manipulable. Seguidamente y con ayuda de la función *read_csv*, guardo los datos en memoria en un vector de estructuras que he denominado *FicheroCSV*. Cada *FicheroCSV* contiene la información de cada línea del fichero *csv* original incluyendo un vector de los datos y un string que indica la clase de los mismos. Por tanto, consideraremos cada conjunto de datos como un *vector<FicheroCSV>*.

Seguidamente, y para aplicar la *5-fold cross validation*, creo un vector con 5 particiones que contienen punteros (para reducir el coste de memoria) a estructuras *FicheroCSV* repartidas de forma equitativa. Los datos se introducen en la partición ya normalizados.

Una vez que tenemos el vector<vector<FicheroCSV*>>, ya podemos repartir los datos en train y test y proceder como hemos descrito anteriormente con los algoritmos programados.

Primero desarrollé una versión del **k-NN** sin leave-one-out, seguidamente programé el algoritmo **Greedy (Relief)** y finalmente la **Búsqueda Local** con la cual tuve que modificar parte del código del **k-NN** para que aplicase el leave-one-out.

En el archivo *LEEME.txt* se encuentra explicado el proceso para replicar las ejecuciones del programa. Así que me limitaré a resumirlo:

Existe un make que genera el ejecutable del programa. El programa ejecutable ha sido compilado con g++ y optimizado con -O2 para reducir los tiempos de ejecución.

./bin/p1 ./data/archivo.csv seed

Ejecuta los tres algoritmos programados en esta práctica sobre el conjunto de datos contenido en el *archivo.csv* (con $archivo = \{colposcopy, ionosphere, texture\}$) usando como semilla el número *seed* pasado como parámetro en el **Búsqueda Local**.

./bin/p1 seed

Crea un archivo *tablas_seed.csv* con los resultados de ejecutar los tres algoritmos implementados sobre los tres conjuntos de datos (*colposcopy.csv*, *ionosphere.csv*, *texture.csv*) y vuelca su contenido en una tabla. El parámetro *seed* indica la iniciacización de la semilla en la **Búsqueda Local**.

Tanto los ficheros .csv como el fichero tablas_seed.csv que se genere se podrán encontrar en el directorio ./data.

6. Experimentos y Análisis de Resultados

Para la obtención de los resultados de esta práctica, hemos utilizado los siguientes conjuntos de datos:

- **Colposcopy**: Conjunto de datos de secuencias colposcópicas extraídas a partir de las características de las imágenes. Contiene 287 ejemplos con 62 características cada uno clasificados en 2 categorías.
- **Ionosphere**: Conjunto de datos de radar que indican el número de electrones libres en la ionosfera partir de las señales procesadas. Contiene 352 ejemplos con 34 características cada uno clasificados en 2 categorías.
- **Texture**: Conjunto de datos para distinguir entre 11 tipos de texturas diferentes a partir de imágenes. Contiene 550 ejemplos con 40 características cada uno clasificados en 11 categorías.

El algoritmo de **Búsqueda Local** depende de un parámetro que especifica la semilla para le generación de números aleatorios. Voy a utilizar para el análisis de estos resultados SEED=23. Los resultados se guardarán por tanto en el archivo *tablas_23.csv* del directorio *data*.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza σ^2 donde $\sigma = 0.3$. Como criterio de parada en la **Búsqueda Local**, se va a usar el número de evaluaciones que

se han realizado así como el número vecinos por característica explorados. En nuestro caso los valores serán 15000 y $20 \cdot tam\_vector\_pesos$ respectivamente.

Las tablas se generan automáticamente con la ejecución del programa y contienen los resultados para cada partición, así como la media de los resultados para cada algoritmo.

A continuación, se muestran las tablas para cada fichero y cada algoritmo utilizado:

	COLPOSCOPY		IONOSPHERE		TEXTURE	
	%tasa_clas	*%tasa_red*	*%tasa_clas*	*%tasa_red*	*%tasa_clas*	*%tasa_red*
1-NN	74,9062	0	86,5996	0	92,5455	0
Greedy-Relief	74,9062	40,9677	87,4567	2,94118	93,0909	5,5
BL	73,8536	81,9355	85,4648	85,8824	90,3636	84,5

Figure 1: Tabla datos comparativa.csv

Analizando los resultados de la tabla, podemos ver que los mejores resultados son obtenidos por el algoritmo de **Greedy-Relief**, igualando al **1-NN** para el conjunto de datos *colposcopy*. Mientras que el **Búsqueda Local** es el que obtiene en los tres casos la mejor tasa de reducción, lo que indica que se necesitan menos atributos para clasificar los datos con respecto al resto de algoritmos.

Podemos ver los resultados totales obtenidos para cada conjunto de datos:

	COLPOSCOPY			
	1-NN			
	%tasa_clas	*%tasa_red*	*Agr,*	*T(seg)*
Partición 1	81,0345	0	40,5172	0,001578
Partición 2	70,6897	0	35,3448	0,001331
Partición 3	77,193	0	38,5965	0,001327
Partición 4	68,4211	0	34,2105	0,001139
Partición 5	77,193	0	38,5965	0,001116
Media:	**74,9062**	**0**	**37,4531**	**0,0012982**
	Greedy-Relief			
Partición 1	79,3103	37,0968	58,2036	0,004498
Partición 2	72,4138	35,4839	53,9488	0,004435
Partición 3	80,7018	35,4839	58,0928	0,004423
Partición 4	70,1754	38,7097	54,4426	0,004431
Partición 5	71,9298	58,0645	64,9972	0,004348
Media:	**74,9062**	**40,9677**	**57,937**	**0,004427**
	BL			
Partición 1	75,8621	75,8065	75,8343	10,0139
Partición 2	75,8621	82,2581	79,0601	11,9539
Partición 3	68,4211	85,4839	76,9525	13,6244
Partición 4	77,193	82,2581	79,7255	14,3424
Partición 5	71,9298	83,871	77,9004	11,8879
Media:	**73,8536**	**81,9355**	**77,8945**	**12,3645**

Figure 2: Tabla datos colposcopy.csv

	IONOSPHERE			
	1-NN			
	%tasa_clas	*%tasa_red*	*Agr,*	*T(seg)*
Partición 1	90,1408	0	45,0704	0,001203
Partición 2	80	0	40	0,000921
Partición 3	82,8571	0	41,4286	0,000963
Partición 4	92,8571	0	46,4286	0,000906
Partición 5	87,1429	0	43,5714	0,000893
Media:	**86,5996**	**0**	**43,2998**	**0,0009772**
	Greedy-Relief			
Partición 1	90,1408	2,94118	46541	0,00387
Partición 2	81,4286	2,94118	42,1849	0,003897
Partición 3	82,8571	2,94118	42,8992	0,004776
Partición 4	92,8571	2,94118	47,8992	0,003898
Partición 5	90	2,94118	46,4706	0,003897
Media:	**87,4567**	**2,94118**	**45199**	**0,0040676**
	BL			
Partición 1	87,3239	85,2941	86,309	2,72671
Partición 2	84,2857	82,3529	83,3193	3,67635
Partición 3	84,2857	85,2941	84,7899	3,15949
Partición 4	88,5714	91,1765	89,8739	2,8825
Partición 5	82,8571	85,2941	84,0756	3,10376
Media:	**85,4648**	**85,8824**	**85,6736**	**3,10976**

Figure 3: Tabla datos ionosphere.csv

	TEXTURE			
	1-NN			
	%tasa_clas	*%tasa_red*	*Agr,*	*T(seg)*
Partición 1	93,6364	0	46,8182	0,003182
Partición 2	89,0909	0	44,5455	0,002859
Partición 3	94,5455	0	47,2727	0,003105
Partición 4	92,7273	0	46,3636	0,002673
Partición 5	92,7273	0	46,3636	0,002717
Media:	**92,5455**	**0**	**46,2727**	**0,0029072**
	Greedy-Relief			
Partición 1	91,8182	15	53,4091	0,011475
Partición 2	91,8182	2,5	47,1591	0,01171
Partición 3	95,4545	2,5	48,9773	0,011494
Partición 4	92,7273	2,5	47,6136	0,012435
Partición 5	93,6364	5	49,3182	0,011078
Media:	**93,0909**	**5,5**	**49,2955**	**0,0116384**
	BL			
Partición 1	90,9091	87,5	89,2045	16,8393
Partición 2	93,6364	80	86,8182	8,03304
Partición 3	89,0909	85	87,0455	11,8779
Partición 4	87,2727	82,5	84,8864	8,92972
Partición 5	90,9091	87,5	89,2045	21,8869
Media:	**90,3636**	**84,5**	**87,4318**	**13,5134**

Figure 4: Tabla datos texture.csv

Vemos que en general el algoritmo que más tarda en ejecutarse es el **Búsqueda Local**, algo evidente, pues es el que más evaluaciones e iteraciones realiza. El conjunto de datos *texture* tarda más pero es el que mayor tasa de clasificación recibe mientras que el segundo con mayor tiempo *colposcopy* aún superando en tiempo de ejecución al de *ionosphere*, este último supera su tasa de clasificación en un 12% e incluso supera la de reducción en la **Búsqueda Local**.

Por ahora no podemos extraer muchas más conclusiones, deberemos esperar a desarrollar el resto de prácticas y algoritmos evolutivos para ver como se comportan estos frente a los conjuntos de datos dados y hacer las comparaciones oportunas..

7. Bibliografía

[1] Guión de Prácticas y Seminarios de la Asignatura de Metaheurísticas.